A comparison of Context-Oriented and Aspect-Oriented Programming

Matthias Springer

Hasso Plattner Institute, Advanced Modularity, seminar paper

Abstract. Aspect-oriented programming and context-oriented programming were designed to modularize cross-cutting concerns that would otherwise lead to scattered code. We illustrate conceptual differences between aspect-oriented programming and context-oriented programming in order to point out which technique should be used for a given use case. Besides, we motivate that pointcuts and immediate layer activation notifications can be useful in context-oriented programming.

1 Software development for mobile devices

In this paper, we will compare aspect-oriented programming and context-oriented programming, two techniques for modularizing cross-cutting concerns. We will show two typical use cases in the area of mobile devices, immediate feedback and power management, that we will implement using aspect-oriented programming and context-oriented programming.

1.1 Immediate feedback

Haptic feedback and auditive feedback are well-known patterns in human-computer interaction [7]. The idea is to provide immediate feedback to user input, such that the user can be certain that the device received the input. In this example the mobile device should vibrate for a quarter of a second after the user touched the touch screen or pressed a keyboard button or another hardware button. Alternatively, the device should play an unobtrusive click sound.

1.2 Power management

Figure 1 shows four power states for a mobile device. If the battery charge level is 50% or greater, then the device is in the high battery state. In that state, no power saving mechanisms are activated, i.e. the device behaves as if no power management would be present.

The medium battery state activates some minor power saving mechanisms, if the battery charge level is lower than 50%. The sound volume is reduced by 50% and the display brightness is dimmed by 50% after five seconds without user input.

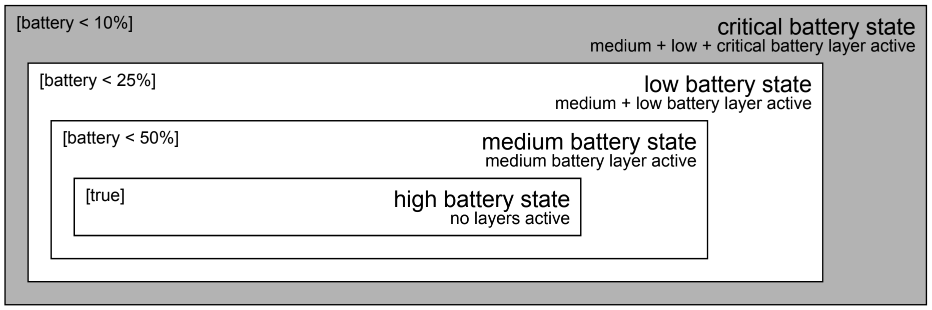

Fig. 1. Layered power states for mobile power management.

If the battery charge level drops below 25%, further battery saving mechanisms from the low battery state are actived, in addition to the already activated mechanisms from the medium power state. The sound volume is reduced by another 50% and the Wifi bandwidth is reduced. Furthermore, the display is turned off after 10 seconds without user input.

The critical power state is activated, if the battery charge level drops below 10%. Power saving mechanisms from the low and the medium battery state remain active. In addition, the critical power state deactivates Wifi entirely and deactivates launching non-critical applications like games and multimedia applications.

2 Previous work

Apel et al. found out that cross-cutting concerns can be classified as homogenous or heterogeneous cross-cutting concerns [1].

Kiczales et al. presented the concept of *aspect-oriented programming* [11] as a means to modularize cross-cutting concerns in object-oriented software components. AspectJ [10] is an aspect-oriented extension of the Java programming language and a fully developed programming language that is widely used in software development and in research. Aspect-oriented extensions exist for most well-established object-oriented programming languages [12].

Hirschfeld et al. proposed a similar concept called *context-oriented programming* [9] for modularizing context-dependent cross-cutting concerns in object-oriented software systems. With JCop, Appeltauer and Hirschfeld provided a context-oriented extension of the Java programming languages. Furthermore, context-oriented extensions exist for Lisp, Smalltalk, Python, Ruby and JavaScript [3].

In Section 3, we will show that immediate feedback is a homogeneous cross-cutting concern and provide an AspectJ implementation. In Section 4, we will show that power management is a heterogeneous cross-cutting concern and provide a JCop implemation. Section 5 will compare both paradigms.

3 Homogeneous cross-cutting concerns

Cross-cutting concerns that add the same behavioral variation at multiple points in the program are called homogeneous [1]. We will now explain, how aspect-oriented programming can be used to modularize homogenous cross-cutting concerns.

3.1 Basic concepts of aspect-oriented programming

With aspect-oriented programming, we can specify that certain code, called advice, should be executed at well-definied points in the execution of the program. These well-defined points are called join points and every implementation of an aspect-oriented programming language has its own join point model, i.e. a definition of where advice code can be executed. We can use pointcut designators to define a set of join points that can be advised by advice. This is useful for homogeneous cross-cutting concerns, where the same piece of advice must be executed at a set of join points. Modularized cross-cutting concerns are called aspects and consist of pointcuts and advice code.

3.2 Example: Immediate feedback for mobile devices

For immediate feedback, the same piece of code, `feedback` in Figure 2, needs to be executed before the methods for handling the respective input events are executed. We therefore call immediate feedback a homogeneous cross-cutting concern.

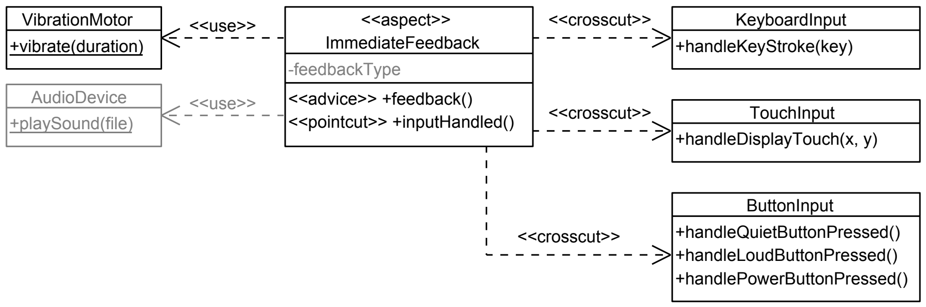

Fig. 2. Aspect-oriented implementation of immediate feedback for input events.

In Figure 2, the pointcut `inputHandled` designates all methods in `KeyboardInput`, `TouchInput` and `ButtonInput` for handling input events, i.e. all methods starting with `handle`.

3.3 Implementation with AspectJ

Listing 3 shows an implementation of immediate feedback for input events with AspectJ. The aspect concentrates all feedback-related code at a single position. In an ordinary Java implementation, code fragements for invoking the advice code would be scattered among all input handler classes.

```
aspect ImmediateFeedback {

  public static enum FeedbackMode {
    NONE, AUDITIVE, HAPTIC
  }

  private FeedbackMode feedbackMode;

  pointcut inputHandled():
    execution(public void handle*(..));

  before(): inputHandled() {
    if (feedbackMode == FeedbackMode.AUDITIVE) {
      AudioDevice.playSoundfile("click.wav");
    }
    else if (feedbackMode == FeedbackMode.HAPTIC) {
      VibrationMotor.vibrate(0.25);
    }
  }

}
```

Fig. 3. Implementation of immediate feedback with AspectJ.

An AspectJ aspect may have fields and methods, implement interfaces and extend other aspects. Therefore, we can use an aspect like a class, except for instantiating it explicitly.

4 Heterogeneous cross-cutting concerns

Cross-cutting concerns that add different behavioral variations at multiple points in the program are called heterogeneous [1]. We will now explain how the concept of context-oriented programming can be used to modularize such cross-cutting concerns.

4.1 Basic concepts of context-oriented programming

Context-oriented programming allows us to specify that certain methods should be supplemented with additional behavior or be replaced entirely, based on contextual information. Contextual information is any data computationally accessible in the program.

We call methods that are modified base methods. Base methods are supplemented by partial method definitions, just as advice code adds additional behavior at various join points in aspect-oriented programming. Partial method definitions are enclosed in layers, just as pointcuts and advice are enclosed in aspects. The runtime environment decides which layers are activated, based on contextual information. Multiple layers may be activated at the same time, thus allowing multiple partial method definitions for the same base method to be active at the same time, just as multiple pieces of advice may advise a single join point.

4.2 Example: Power management for mobile devices

Power management is mostly a heterogeneous cross-cutting concerns, because multiple base methods are supplemented with different behavior. For example, in Figure 4, `Shell.runApplication(file)` and `AudioDevice.write(data, volume)` are supplemented with different partial methods.

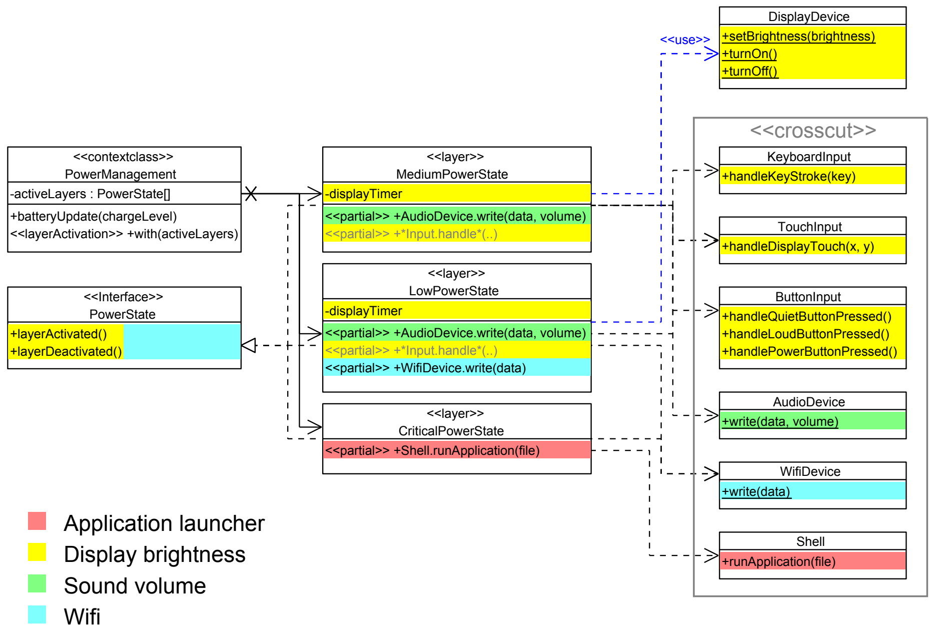
DisplayDevice
+setBrightness(brightness)
+turnOn()
+turnOff()
<<use>>
<<crosscut>>
KeyboardInput
+handleKeyStroke(key)
TouchInput
+handleDisplayTouch(x, y)
ButtonInput
+handleQuietButtonPressed()
+handleLoudButtonPressed()
+handlePowerButtonPressed()
AudioDevice
+write(data, volume)
WifiDevice
+write(data)
Shell
+runApplication(file)
<<contextclass>>
PowerManagement
-activeLayers : PowerState[]
+batteryUpdate(chargeLevel)
<<layerActivation>> +with(activeLayers)
<<Interface>>
PowerState
+layerActivated()
+layerDeactivated()
<<layer>>
MediumPowerState
-displayTimer
<<partial>> +AudioDevice.write(data, volume)
<<partial>> +*Input.handle*(..)
<<layer>>
LowPowerState
-displayTimer
<<partial>> +AudioDevice.write(data, volume)
<<partial>> +*Input.handle*(..)
<<partial>> +WifiDevice.write(data)
<<layer>>
CriticalPowerState
<<partial>> +Shell.runApplication(file)
Application launcher
Display brightness
Sound volume
Wifi

Fig. 4. Context-oriented implementation of power management for mobile devices.

However, all methods for input handling starting with `handle` share the same partial method. Therefore, we can consider the display brightness control part in power management a homogeneous cross-cutting concern. Having said that, we may at the same time consider it a heterogenous cross-cutting concern as different partial method combinations are applied, based on the current battery charge level.

4.3 Implementation with JCop

Context evaluation and layer activation. Listing 7 shows the context class `PowerManagement` that is responsible for activating and deactivating layers. `PowerManagement` observes the battery and receives updates on charge level changes. These updates are needed only for layer-specific actions on layer activation or deactivation, i.e. when the layer activation condition is satisfied or unsatisfied for the first time since the last deactivation or activation. For instance, `CriticalPowerState` must activate or deactivate the Wifi in `layerActivated` or `layerDeactivated`.

If the power states consisted of partial method definitions only, then we would need neither the observer nor the `activeLayers` field nor the `PowerState` interface. We could then implement layer (de)activation using `when-with` statements only.

Power state implementation as a layer. Listing 8 shows the low power state layer. The thread `displayTimer` is activated on layer activation and when the mobile device receives input from the touch display, the keyboard or a hardware button[1]. At first the display is undimmed, then the five seconds dim timer is reset. Futhermore, `MediumPowerState` contains a partial method definiton for `AudioDevice.write` that lowers the volume parameter by 50%.

4.4 Implementation issues: Initial layer activation and deactivation

Context-oriented programming provides a mechanism to supplement base methods with additional behavior if a certain layer is activated. Sometimes this is not enough. In the previous example we discovered cases where some code had to be executed immediately after a layer was activated or deactivated.

For instance, immediately after the mobile device enters the low power state, the display dim timer has to start, even if the user does not make any input. Similarly, the timer must be stopped and the display brightness restored if the mobile device leaves the low power state.

Wifi deactivation in the critical power state is another example that is triggered merely on layer activation or deactivation.

So far, neither the paradigm of context-oriented programming nor ContextJ or JCop provide a concept that simplifies code execution on layer (de)activation. In Listing 7 we simulated this behavior by observing the battery[2] and calling `layerActivated` and `layerDeactivated` on layer recomposition.

[1] Only the partial method for `handleDisplayTouch` in shown in Listing 8 but identical partial methods exist for the other four input handlers.

[2] The current JCop implementation does not allow context classes to implement interfaces, thus in reality requiring the Java Reflection API.

A context-oriented programming language could monitor layer activation conditions the whole time and call `layerActivated` and `layerDeactivated` on its own if these methods are provided by the programmer. Monitoring activation conditions can be done by polling the condition repeatedly or by providing an observer interface. The implementation of polling is trivial but it can cost too much performance.

```
contextclass PowerManagement implements Observer<float> {

  when (updateValue() < 0.50f): with(MediumPowerState);

  when (updateValue() < 0.25f): with(LowPowerState);

  when (updateValue() < 0.10f): with(CriticalPowerState);

}
```

Fig. 5. JCop observer syntax proposition.

Listing 5 shows how an observer concept provided by the programming language could look like on the programmer's side. `Observer` serves as a marker interface and provides the generic type of the observer update parameter. `updateValue` returns the (cached) value from the last observer update. When an update is triggered layers are automatically notified about their (de)activation, exactly as in Listing 7, but with less code.

5 Conceptual and implementational differences

In this section, we compare aspect-oriented programming and context-oriented programming and their implementations AspectJ and JCop.

5.1 Homogeneous and heterogeneous cross-cutting concerns

Hirschfeld et al. mention that context-oriented programming aims at modularizing heterogeneous cross-cutting concerns. This is illustrated by the fact that context-oriented programming has no concept of pointcuts and that partial methods always extend only one base method.

Aspect-oriented programming, in contrast, allows advice code to be executed at multiple join points, thus allowing the modularization of both types of cross-cutting concerns.

Most context-depending cross-cutting behavior variations in Figure 4 are heterogeneous. However, in Section 4.2, we realized that diplay brightness control can to some degree be considered a homogenous cross-cutting concern, because the partial method for dimming or turning on/off the display must be applied to multiple base methods. In that case, we would benefit from a pointcut designator, as we would not have to repeat multiple partial method definitions with the same code.

Apart from the join point model in aspect-oriented programming, such a conceptual change would, to some extent, make context-oriented programming an extension of aspect-oriented programming. We can then map the basic concepts of aspect-oriented programming directly to concepts of context-oriented programming[3]. However, this is not true for more advanced concepts like inter-type declarations and `declare` statements in AspectJ.

5.2 Inter-type declarations

Aspect-oriented programming offers advanced mechanisms to change a module's structure. Inter-type declarations in AspectJ are used to add new methods to classes or to change the class hierarchy. With the `declare parents` statement, we can define that a class implements a specific interface or extends a specific class[4].

Context-oriented programming provides no mechanisms of changing the class hierarchy or adding new methods. Partial method definitions in JCop (Listing 6) might sytactically look similar to inter-type declaration for adding methods in AspectJ, but they can in fact only add new behavior to already existing methods.

```
layer LowPowerState {

  /* ... */

  public void AudioDevice.write(byte[] data, float volume) {
    proceed(data, 0.5f * volume);
  }

}
```

Fig. 6. JCop partial method definition.

5.3 Activation of behavioral variations

The term *context-oriented programming* is derived from the idea that contextual information decides whether a layer is activated or not. The way layers are (de)activated varies strongly among context-oriented programming languages.

[3] Partial method definition → advice, join point model → reduced join point model (execution only), pointcut → pointcut, aspect → layer.

[4] There are certain limitations if the class already has a superclass. The new superclass must have the original superclass in its inheritance hierarchy.

The predecessor of JCop, ContextJ [4], supported context (de)activation through `with` and `without` statements only. JCop adds a declarative syntax for layer (de)activation [5]. Basically, every time a method is invoked, JCop first computes all activated layers by evaluating all `when-with` statements [2]. These statements typically fall back on fields or methods of the context class, that provide and interprete the relevant contextual information. Afterwards, partial methods and base method are executed.

In contrast, AspectJ weaves in aspects at compile time [8], making dynamic aspect activation impossible. CaesarJ supports aspect activation at runtime [6]. But CaesarJ still does not feature a concept similar to context classes or declarative layer activation, making it difficult to coordinate layer (de)activation.

Therefore, context-oriented programming fits best for use cases where some behavioral variations must be chosen out of many possible variations, based on contextual information. Aspect-oriented programming fits best for static behavioral variations that are usually not activated or deactivated at runtime.

5.4 Join point model

The join point model in aspect-oriented programming languages defines at which points in the execution of the program advice code can be weaved in. The concrete join point model depends on the implementation, but most implemenations support at least method execution and method call[5]. AspectJ support more join points like constructor execution and field access [10].

In Section 5.1, we tried to map concepts of aspect-oriented programming to concepts of context-oriented programming and noticed that context-oriented programming provides only a very restricted *join point model*. The only supported join point is method execution. This fits to the basic idea of context-oriented programming that certain classes behave differently under certain circumstances, by supplementing base methods (method execution) with additional behavior.

If we need a more sophisticated join point model, e.g. in order to change the constructor or control field access, we therefore should use aspect-oriented programming. Such behavioral variations tend to be non-dependent on contextual information but to be rather static, anyway.

6 Conclusion

In the previous sections we found out that aspect-oriented programming is suitable for modularizing homogeneous cross-cutting concerns that are more or less static at runtime, i.e. no or only little aspect activation or deactivation is dynamically done at runtime. Furthermore, aspect-oriented programming languages typically feature a rich join point model and provide advanced mechanisms to change a module's structure.

[5] At that point, information about the sender is still available.

Context-oriented programming, on the other hand, aims primarily at modularizing heterogeneous cross-cutting concerns with behavioral variations that are composed dynamically at runtime. We showed, though, that in some use cases we would benefit from pointcut designators. Such concepts might be addressed in future versions of context-oriented implementations.

References

1. Sven Apel, Don Batory, and Marko Rosenmller. On the structure of crosscutting concerns: Using aspects or collaborations. In *In Workshop on Aspect-Oriented Product Line Engineering*, 2006.
2. Malte Appeltauer and Robert Hirschfeld. The jcop language specification. *Technische Berichte Nr. 59*, 2012.
3. Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6, New York, NY, USA, 2009. ACM.
4. Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. Contextj: Context-oriented programming with java. *Journal of the Japan Society for Software Science and Technology (JSSST) on Computer Software*, 28(1):272–292, 2011.
5. Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Proceedings of the 9th international conference on Software composition*, SC'10, pages 50–65, Berlin, Heidelberg, 2010. Springer-Verlag.
6. Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. *T. Aspect-Oriented Software Development I*, pages 135–173, 2006.
7. Stephanie Foehrenbach, Werner A. Knig, Jens Gerken, and Harald Reiterer. Natural interaction with hand gestures and tactile feedback for large, high-res displays. In *MITH 08: Workshop on Multimodal Interaction Through Haptic Feedback, held in conjunction with AVI 08: International Working Conference on Advanced Visual Interfaces*, May 2008. Workshop Research Paper.
8. Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD*, pages 26–35, 2004.
9. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.
10. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.
11. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.
12. Luis Miguel Lourenco, Nuno Cerqueira, Pedro Corte-Real, and Rui Barbosa. Aspect oriented programming comparing programming languages. 2005.

A Appendix

```
contextclass PowerManagement implements BatteryObserver {

  private PowerState[] activeLayers;

  private PowerState criticalState = new CriticalPowerState();
  private PowerState lowState = new LowPowerState();
  private PowerState mediumState = new MediumPowerState();

  public void batteryUpdate(float chargeLevel) {
    PowerState[] newLayers = [];

    if (chargeLevel < 0.50f)
      newLayers += mediumState;

    if (chargeLevel < 0.25f)
      newLayers += lowState;

    if (chargeLevel < 0.10f)
      newLayers += criticalState;

    for (PowerState layer : activeLayers - newLayers)
      layer.layerDeactivated();

    for (PowerState layer: newLayers - activeLayers)
      layer.layerActivated();

    activeLayers = newLayers;
  }

  when(true): with(activeLayers);

}
```

Fig. 7. JCop context class for power state (de)activation. This is no valid Java code. To improve readability, Java collection handling was substituted by a pseudo-code-like syntax.

```
layer MediumPowerState implements PowerState {

  private boolean isDimmed = false;

  private Thread displayTimer = {
    sleep(5000);
    DisplayDevice.setBrightness(0.5f);
    isDimmed = true;
  };

  private void unDim() {
    if (isDimmed) {
      DisplayDevice.setBrightness(2f);
      isDimmed = false;
    }
  }

  /* same for KeyboardInput.handleKeyStroke(..), ButtonInput.
      handle*(..) */
  before public void TouchInput.handleDisplayTouch(int x, int
      y) {
    thislayer.unDim();
    thislayer.displayTimer.stop();
    thislayer.displayTimer.start();
  }

  public void layerActivated() {
    displayTimer.start();
  }

  public void layerDeactivated() {
    unDim();
    displayTimer.stop();
  }

  public void AudioDevice.write(byte[] data, float volume) {
    proceed(data, 0.5f * volume);
  }

}
```

Fig. 8. JCop low power state implementation. This is no valid Java code. Thread handling was simplified to improve readability.